Explanation-Based Constraint Programming for a University Timetabling Problem

Chia-Lin Hsieh
Department of Industrial Management,
Aletheia University, Tamsui, Taipei, Taiwan.
hsiehcl@email.au.edu.tw

ABSTRACT- *Explanation-based constraint programming is a new way of solving constraint systems. It allows to propagate constraints of the problem, learning from failure and from the solver and finally allows to get rid of backtrack-based complete search by allowing more free moves in the search space. In this paper, we present our experience in using explanations within constraint programming: how to implement an explanation system, what to use explanation for solving a university timetabling problem. Beside classical uses, we are attempting to solve the problem with the class library of ILOG Solver [7] which leads to a new kind of explanation-based constraint programming.*

Keywords: Explanation, Constraint programming, Timetabling, Contradiction handling

1. Introduction

Constraint programming has been proved extremely successful for modeling and solving combinatorial search problems appearing in fields such as scheduling resource allocation and timetabling. Several languages and systems such as ILOG Solver [7], CHOCO [9] have been developed and widely spread. But these systems are helpless when the constraint system to be solved has no solution. Indeed, only a *no solution* message is sent to the user who is left alone to find: why the problem has no solution, which constraint to relax in order to restore the failure *etc*. These questions yield two fundamental tasks: *identification of constraints to be relaxed* and *efficient constraint suppression*.

Our previous work [6] presented an explanation -based constraint system, where a promising technique using *explanations* provides useful information. The prototype system has been instantiated and evaluated successfully for finite domain constraint satisfaction problems. However, it is not practical enough. It is concerned to experiment with more real-life problems to evaluate the system.

In this paper, we propose a prototype of an explanation-based constraint programming system for a timetabling problem. Timetabling is a process of assigning events or activities to resources such as timeslots, rooms and lecturers which could satisfy all required (or hard) constraints and also preference (or soft) constraints as acceptable as possible. The problem is combinatorial and dynamic. A good and efficient timetabling system is required to manage the rapidly growing academic activities within a limited time and limited room resources. We are attempting to solve the problem with the class library of ILOG Solver [7] which leads to a new kind of explanation-based constraint programming.

2. Explanations within Constraint Programming

We consider here a constraint satisfaction problem (CSP). Decisions are made during variable assignments correspond to adding or removing constraints from the current constraint system. These constraints are called *decision variables*.

2.1. Contradiction explanations

Let us consider a constraint system whose current state is contradictory. A *contradiction explanation* (*a.k.a nogood* [10]) is a subset of the current constraint system of the problem that leads to a contradiction. A contradiction explanation divided into two parts: a subset of the original sets of constraints ($C' \subset C$ in equation (1)) and a subset of decision constraints introduced in the search so far.

$$C \mapsto (C' \wedge v_1 = a_1 \wedge .. \wedge v_k = a_k) \qquad (1)$$

In a contradiction explanation composed of at least one decision constraint, a constraint v_j is rewritten in the following way:

$$C \mapsto (C' \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j \qquad (2)$$

The left hand side of the implication is called an *eliminating explanation* because it justifies the removal of value a_j from the domain of the variable v_j and is noted: $\exp(v_j \neq a_j)$.

Classical CSP solvers use domain reduction techniques. Recording eliminating explanation is sufficient to compute contradiction explanations. Indeed, a contradiction is identified when the domain of a variable v_j is emptied. A *contradiction explanation*[1] can be easily computed with the eliminating explanations associated with each removed value:

$$C \mapsto \neg(\bigwedge_{a \in d(v)} \exp(v \neq a)) \qquad (3)$$

There exist generally several eliminating explanations for the removal of a given value. Recording all of them leads to an exponential space complexity. Another technique relies on forgetting eliminating explanations that are no longer *relevant*[2] to the current variable assignment. By doing so, the space complexity remains polynomial. We keep only one explanation at a time for a value removal.

2.2. Computing explanations

Minimal (*w.r.t.* inclusion) explanations are the most interesting events. Such explanations allow highly focused information about dependency relations between constraints and variables [5]. Unfortunately, computing such an explanation can be exponentially time-consuming [8]. A good compromise between preciseness and easy computation is to use the knowledge embedded inside the constraint solver to provide explanations. Indeed, constraint solvers always know why they remove values from the domains of considered variables. By explicitly stating such information, quite precise and interesting explanations can be computed. To achieve this behavior, it is necessary to alter the code of the solver itself.

The constraint solver that we develop uses an event-based model. During propagation, constraints are awakened each time when a variable domain is reduced (the reduction is an event) and possibly generating new events (value removals). In such a model, a constraint is fully characterized by its behavior regarding the basic events: value removal from the domain of a variable (method **awakeOnRem**), domain bound updates (method **awakeOnInf** and **awakeOnSup**) and a variable instantiation (method **awakeOnInst**)

Example1: (Constraint $x \geq y + c$)

This is one of the basic constraint in our system. It is represented by the **BGT/BGE** class. If the upper bound of x is modified, the upper bound of y

[1] If contradiction explanation doesn't contain any decision constraints, the associated problem is *over-constrained*.

[2] A no-good is said to be *relevant* if all the decision constraints in it are still valid in the current search state [1].

should be lowered to the new value of the upper bound of x taking into account of the constant c. This is coded as:.

```
[awakeOnSup(c:BGT/BGExyc, idx:integer) : void
->if (idx=1) updateSup(c.v2, c.v1.sup - c.cst)]
```

In Example1, **idx** is the index of the variable of the constraint whose bound has been modified. This constraint only reacts to modification of the upper bound on variable x (*c.v1* in the constraint). The method **updateSup** only modifies the value of y (*c.v2* in the constraint). Explanations for events need to be computed when the events are generated, *i.e.* within the propagation code of the constraints. In order to make it as simple as possible, one only needs to add an extra information to the **updateInf** and **updateSup** calls: the actual explanation.

Let us consider Example 1, modifications to be made are quite simple. Indeed, all the information is at hand in the **awakeOnSup** method**.** The modification of the upper bound of variable *c.v2* (or y) is due to the use of constraint itself and the previous modification of the upper bound of variable *c.v1* (or x). An explanation for the modification can be computed using the **becauseOf** method. The source code is then modified in the following way:

```
[awakeOnSup(BGT/BGExyc, idx:integer) : void
  if (idx=1) updateSup(c.v2, c.v1.sup - c.cst,
      becauseOf(c, theSup(c.v1)))]
```

Our implementation of explanations provides a set of tools in order to ease the modification process. The **Explanation** class that captures contradiction and the modification of the domain update method in order to efficiently store the explanations associated to a given variable. These modifications added to each propagation method efficiently construct an explanation-based constraint solver.

3. Using Explanations

Explanations can be used to determine direct or indirect effects of a given constraint on the domain of variables of the problem. But what is interesting in the context is the ability of explanation system.

3.1. Explanations for constraint retraction

A constraint, in an explanation-based constraint system, includes *value removal* and *value restorations.* It is time to see how explanation is useful when dealing constraint retraction. Constraint retraction in dynamic problem has been studied [2], but here we simplify the algorithm due to explanations. When using an explanation-based system, constraint retraction of a given constraint c can be achieved in two main steps:

– *Setting values back*: Setting back values refers to undo the past effects of the constraints. That is, all the associated events which are no more valid should be put back to their respective domain. This step is quite easy by considering all explanations containing the removed constraint.

– *Re-achieving consistency*: A consistency check should be done in order to get a consistent state as if the removed constraints never appeared in the constraint system. Those new removals need to be propagating again. At the end of this process, the system should be in a consistent state. This process is like the ones in [2][5], but we don't need to compute the past effects of a constraint since each explanation in our system contains all the information at once. We just need to compute the set of explanations containing the retracted constraint.

3.2. Explanations for contradiction handling

Explanation can be used to select and relax a constraint which allows the discovery of new solutions. Once a contradiction occurs, there is no need to backtrack: simply consider the explanation that justifies the lack of more solutions for the current problem (calling **explain**(FailingVariable(), domain, e)), and select a constraint in it. As shown in [4], one needs to select the more recent constraint in the explanation in order to remain complete. In order to move from the dead-end, one can remove the considered constraint and add its negation. Figure 1 shows such a contradiction handling mechanism.

```
handleContradiction(): void
 {
  if FailingVariable?()       // a failure occurred
   {
  let e =conflict_set();
    {
       explain(FailingVariable(), domain, e);  //compute an explanation
       if empty(e)                      // the problem unsolvable
          contradiction!();         // raise a contradiction
        else  {
            let sc = select Constraint(e);
          if known?(sc)
          {    unassignedVars= add(sc.v1);
               remove (sc);    //relax the constraint and remove its effects

             e =delete(sc);      //posting the associated negation constraint
            post(current_pb, e);
            propagate(current_pb);  //restoring consistency

            if  contradiction!()
                handleContradiction();   //doing it recursively }
 }}}}
```

Figure 1. The code for contradiction handling

4. Application to Timetabling Problem

In this section, explanation-based constraint programming is used to solve a university timetabling problem. The objective is to implement a prototype for efficient and comfortable timetabling in our own department. In addition, we expect a significant improvement of the timetable's "quality" and an acceleration of the generation process. The system will also be designed to be adaptable to the requirements of other departments. First prototype has currently being validated in our department.

4.1 Overview of the prototype timetabling system

The prototype of our timetable system adopts an object-oriented approach [3] which separates problem specifications and constraint solver in two different layers to enhance model formulation and maintenance (see Figure 2). The layer of the problem specifications is consisting of three modules (*e.g.* Constraint Manager, Timetable Generator and Optimization Criteria) which can be modeled according to the user's requirements. In fact, the constraint solver provides tools for explanation-based search paradigm, contradiction handling mechanism and constraint posting methods. Whatever changes take place, we may only change the problem specifications (*e.g.* constraints, variables, *etc.*) without disturbing the logic of the constraint solver.

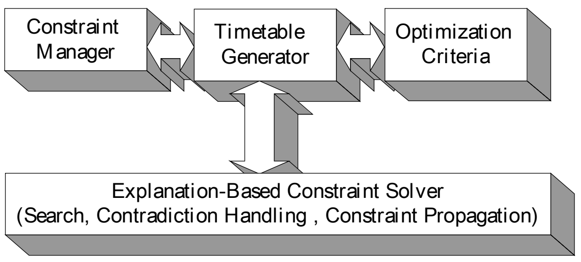
Constraint Manager
Timetable Generator
Optimization Criteria
Explanation-Based Constraint Solver
(Search, Contradiction Handling , Constraint Propagation)

Figure 2. Prototype for timetabling problem

4.2 Design results

Firstly, in Constraint Manger module, we build a first library of constraints to organize the variable limitations in requirement classes called *constraint levels*. A constraint level is just a container for concrete constraints. For example, all constraints of the type "no concurrence of courses of the same lecturer" are put on one constraint level. Each constraint of this level consists of all the courses of one lecturer and the level comprises time preferences of each lecturer.

Constraints are divided into two groups: *hard* and *soft* constraints. Hard constraints are the minimum requirements to be satisfied, otherwise it is impossible to generate a reasonable timetable. On the other hand, soft constraints should be satisfied as acceptable as possible. In our prototype, we arrange the constraint levels by assigning to each of them a priority due to their subjective importance. The result is a list of all constraint levels by priority. The priority 0 is assigned to the level of all hard constraints. Thus, the objective of the timetabling problem is to satisfy as many constraint levels as possible.

All concrete constraints with similar meaning can be abstracted to constraint classes. For example, all possible types of "no concurrence of courses" can be abstracted to be the class called "*NotConcurrence*" which is the most important constraint class. Figure 3 shows an extract of Booch's [3] class diagram as a result of the analysis and design process.

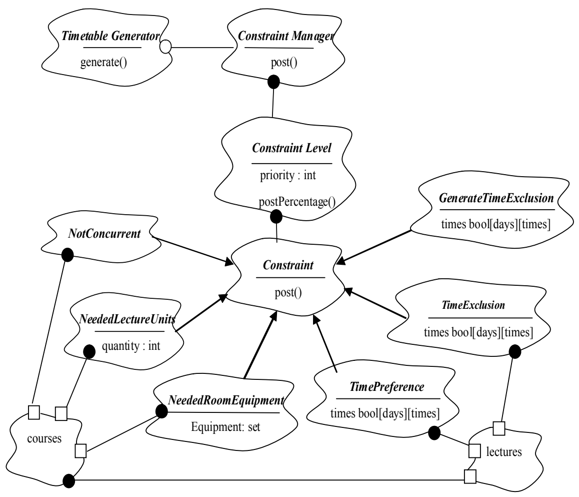

Figure 3. Class diagram of constraint manger

It is shown in Figure 3 that all constraint levels containing all concrete constraints are collected in one object of the class "*Constraint Manger*" which can be accessed by the object of the class "*Timetable Generator*". The diagram was reduced to those classes related to constraints and thus relevant to Constraint Solver. As one can see all constraints classes (*e.g "NotConcurrent",* "*TimePreference*", "*GenerateTimeExclusion*", *etc.*) are derived from an abstract base class called "*Constraint*". This base class only consists of the virtual operation "*post*" that has to be inherited to each class. It makes sure that one object from each constraint class can convert its data member to constraints so that Constraint Solver can understand.

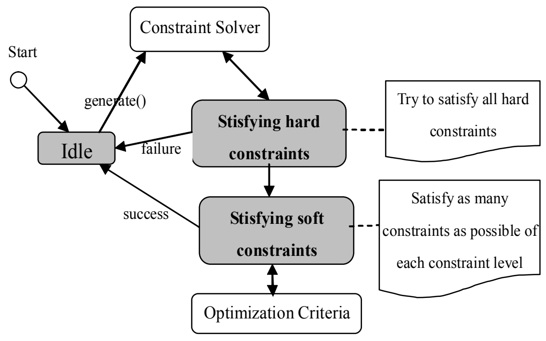
Constraint Solver
Start
generate()
Idle
failure
Stisfying hard constraints
Try to satisfy all hard constraints
success
Stisfying soft constraints
Satisfy as many constraints as possible of each constraint level
Optimization Criteria

Figure 4. State diagram of timetable generator

Secondly, in Timetable Generator module, we design the class "Timetable Generator" (see Figure 4). When the timetable generator gets the message to generate a timetable, it consults the constraint solver to search for solutions satisfying all hard constraints. If there exists no solution of these constraints, the generation has failed as they are the minimum requirements to the timetable.

If all hard constraints could be satisfied, the generator proceeds to the state "Satisfying soft constraints". Thus, the objective is to satisfy as acceptable as possible of each constraint level. In this state, the generator inquires optimization criteria from the user to receive an optimal solution when all constraints of each level are satisfied. However, these optimization criteria are user-defined. The generator leaves this state always with a "success" result meaning that a valid and so-called "best" timetable *(w.r.t.* optimization criteria) could be generated as at least all hard constraint are satisfied.

5. First Results

The proposed system is in fact mad of several modules delivered in the class library of ILOG Solver [7] and modified with C++ which is dedicated to explanation-based constraint programming. It is implemented using real university timetabling data of our department. We were very quickly acquainted with ILOG Solver and wrote our first results. It is striking the very readable and compact codes of the system. Simple models can easily be extended and the Solver is very useful for rapid prototyping. In fact, we are not the first ones attempting to solve timetabling problem in our university. A lot of academic staff have tried but failed. With constraint solver as part of our prototype, we have already achieved the results as the conventional planners in our department, but in significantly less time.

Our results demonstrate the system performance where the feasible and acceptable solution is found in a reasonable time compared to the size and the complexity of the problem with the help of the explanation-based constraint solver. It successfully finds the acceptable solution. Due to these facts, our system behaves desirable features.

6. Conclusion

We presented an original use of explanation-based constraint programming to propose a solution for a difficult timetabling problem. A library of dedicated constraints is developed to solve this problem. Explanations are also used to provide an efficient system: conflicts are identified and explained, the search can be completely driven automatically or by the user. The system is implemented using real university data. The solution is found in a reasonable time. The first results show flexibility and adaptability of the tool with the help of object-oriented design of the system. In the future, the proposed prototype will try for further improvement. The aim is to apply explanation-based constraint programming to develop an efficient and appropriate university-wide professional version.

Reference

[1] R. J. Bayardo, and D. P. Miranker, "A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem", AAAI-96, pp. 298-304, 1996.

[2] C., Bessiere, "Arc-consistency in dynamic constraint satisfaction problems", Proceedings of the 9th National Conference on Artificial Intelligence, pp. 221-226, 1991.

[3] G. Booch, Object-oriented analysis and design, Benjamin/Cummings,1994.

[4] M. L. Ginsberg, "Dynamic backtracking", Journal of Artificial Intelligence Research, vol. 1, pp. 25-46, 1993.

[5] C. L. Hsieh and J. Archibald, "A dependency -based constraint relaxation scheme for over -constrained problems", International Computer Symposium, pp. 134-141, 1998.

[6] C. L. Hsieh, "Conflict resolution with explanations and best-first search for solving over-constrained problems", Proceedings of 5th Conference on Artificial Intelligence and Applications, pp. 429-435, 2000.

[7] ILOG, Ilog Solver reference manual, 2001.

[8] U. Junker, "QUICKXPLAIN: conflict detection for arbitrary constraint propagation algorithms", IJCAI'01 Workshop on Modeling and Solving Problems with Constraints, 2001.

[9] F. Laburthe, "Choco: implementing a CP Kernel", CP'2000 Post Conference Workshop on Techniques for Implementing Constraint Programming Systems, 2000.

[10] T. Schiex and G. Verfaillie, "Nogood recording for static and dynamic constraint satisfaction problems", International Journal of Artificial Intelligence Tools", vol. 3, no. 2, pp. 187-207, 1994.